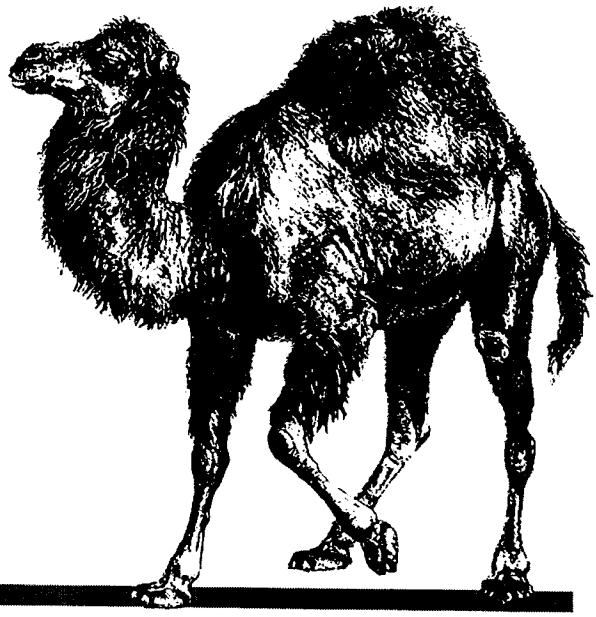
UNIX Programming

Perlプログラミング



Programming

perl

Larry Wall and Randal L. Schwartz 著
近藤 嘉雪 訳

A NUTSHELL HANDBOOK

ソフトバンク株式会社

シェルでは，シングルクォートの内側以外のあらゆる場所で変数展開が行われるが，Perl では，定められた場所――ダブルクォートの内側など――でしか変数展開（および \n などの逆スラッシュの解釈）は行われない。バッククォートで囲んだ文字列でも，ダブルクォートのときと同様に変数展開（と逆スラッシュの解釈）が行われるが，バッククォートの場合には，変数展開で得られた文字列をさらにコマンドとして実行して，その結果得られた出力が最終的な値となる：

```
$lines = `wc -l $filename`;
```

（**注意**：入力や出力をリダイレクトする必要がなく，ただコマンドを実行したいだけなら，system 関数を使うこと。第 4 章の system() 関数〔260ページ〕を見よ。）

また，次節で紹介するパターンマッチでも，変数展開が行われる。

変数名の直後の文字が，変数名の続きに見えるおそれがあるときには，変数名をブレースで囲む必要がある：

```
/the merry ${mo}nth of May/;      # 変数 $mo が展開され，
                                  # nth はそのまま残される。
```

この例で，ブレースがないと変数 $month の値が展開されてしまう。

1.4■パターンマッチ

Perl には，m// と s/// の 2 種類の**パターンマッチ演算子**（pattern matching operator）がある。これらは，それぞれ**マッチ演算子**（match operator），**置換演算子**（substitution operator）と呼ばれる（スラッシュの代わりに，英数字以外の任意の文字を使うことができる）。スラッシュをデリミタに使うときには，m// 演算子の m は省くことができる。そのために，m// 演算子は，単に // 演算子と呼ばれることもある（UNIX プログラムの多くでは，デリミタとしてスラッシュしか使えないのも，その理由の 1 つである）。m// 演算子は，デリミタ間に指定されたパターンを文字列の中から探して，マッチした（パターンを含んでいる）かどうかによって，真または偽を返す。s/// 演算子も同じ処理を行うが，文字列のうちマッチした部分を，2 番目と 3 番目のデリミタの間に指定されているもので置き換える。

パターンは，**正規表現**（regular expression）を使って指定する。おそらく読者の多くは，*sed* や *vi* などの UNIX ツールを通じて，すでに正規表現に親しんでいることだろう。しかし Perl が扱う正規表現は，他の正規表現ほとんどのスーパーセットになっているので，いずれにせよ次の一覧表を注意深く見てほしい：

.	改行以外の任意の 1 文字にマッチ
[a-z0-9]	集合に含まれるいずれか 1 文字にマッチ
[^a-z0-9]	集合に含まれない文字 1 つにマッチ
\d	数字 1 文字にマッチ。[0-9] と同じ
\D	数字以外の 1 文字にマッチ。[^0-9] と同じ
\w	英数字（alphanumeric：単語を構成する文字）の 1 文字にマッチ。[a-zA-Z0-9_] と同じ
\W	英数字以外の 1 文字にマッチ。[^a-zA-Z0-9_] と同じ
\s	空白文字（スペース，タブ，改行など）の 1 文字にマッチ
\S	空白文字以外の 1 文字にマッチ
\n	改行文字（newline）にマッチ
\r	復帰文字（return）にマッチ
\t	タブ（tab）にマッチ
\f	改ページ文字（form feed）にマッチ
\b	バックスペース（backspace）にマッチ（[] の内側のみ）
\0	ヌル文字（null character）にマッチ
\000	これもヌル文字にマッチする。なぜなら……
nnn	8 進値 *nnn* の ASCII 文字にマッチ
\x*nn*	16 進値 *nn* の ASCII 文字にマッチ
\c*X*	ASCII コントロール文字にマッチ
metachar	その文字自身にマッチ（|、\.、* など）
(abc)	後方参照するためにマッチを記録しておく
\1	最初のカッコがマッチしたものにマッチ
\2	2 番目のカッコがマッチしたものにマッチ
\3	以下同様……
x?	0 または 1 個の *x* にマッチ。*x* は上記のいずれか

x*	0個以上のxにマッチ
x+	1個以上のxにマッチ
$x\{m,n\}$	m個以上n個以下のxにマッチ
abc	a, b, cにこの順番でマッチ
fee|fie|foe	fee, fie, foeのどれか1つにマッチ
\b	単語の境界にマッチ（[]の外側のみ）
\B	単語の境界以外にマッチ
^	行または文字列の先頭にマッチ
$	行または文字列の末尾にマッチ

他の正規表現とは違って，Perlの正規表現では，メタキャラクタとして使うときには，カッコ，ブラケット，縦棒の前には逆スラッシュを置かない。反対に，普通の文字として使うときには，これらの直前に逆スラッシュを置かなければならない。（このような逆スラッシュの扱いには，それなりの理由がある。これらの文字はほとんどの場合メタキャラクタとして使われるので，もしメタキャラクタとして使うときに逆スラッシュが必要だとすると，プログラムが逆スラッシュ中毒（backslashitis）になってしまうだろう。また，この後すぐお話しするように，メタキャラクタを簡単にクォートできるという利点もある。）このほかにも，数字，英数字，空白文字を表す略記法が用意されている。

他の正規表現と同じように，パターンがカッコを含んでいるとき，それより後ろの部分では，カッコの対にマッチしたものを\1，\2，\3……によって参照することができる（数字は，パターンの左端から数えて，何個目の左カッコかを示す）。\1，\2……のことを**後方参照**（backreference）と呼ぶ。後方参照は，カッコで囲まれたパターンではなく，カッコが実際にマッチしたものを表すことに注意しよう。後方参照は，まったく同じ文字の並びにマッチしなければならない。10ページで取り上げたbaregrepプログラムを，次のように起動したとしよう：

```
baregrep '(\w+)\s*=\s*\1' *.c
```

この結果，Cソースファイルのうち，hump = humpのように，変数に自分自身を代入している行がすべて表示される。（ところで，この例ではhum = umpのような行も表示されてしまう。なぜだろうか？　**ヒント**：\bは単語の境界にマッチする。）

マッチした文字列のうち，カッコで囲まれた部分を**部分文字列**[15]（substring）という。多くの言語では，パターンや置き換え文字列の中で部分文字列を後方参照することができるが，Perlはさらに一歩先をいっている。パターンマッチが成功すると，すべての部分文字列は特殊変数$1，$2，$3……に残されるようになっている。これらの変数には，後方参照と同じ番号がつけられている。後方参照よりも，$1のような変数を使うことが推奨される。（パターン自身の中は別である。なぜなら，パターンマッチの最中には，これらの変数はまだ作られていないから。）したがって，置換演算子を例にとると，置き換え文字列の中では\1，\2の代わりに，$1，$2を使うほうがよい。

ところで前節では，ダブルクォートで囲んだ文字列と同じように，パターンも変数展開の対象になると説明した。パターンの中では，$は，行末（または文字列の末尾）を表さない場合に限って，変数展開の対象になる：

```
/the merry month of $mo/;    # 変数$moの値を展開する。
/the merry month of May$/;   # $は行末にマッチする。
```

同様に，置換演算子では，パターン，置き換え文字列ともに変数展開の対象になる：

```
s/$monthname/$monthnum/;     # April を 4 へ置換，といったことをする。
s/\r\n$/\n/;                 # 行末の CRLF を LF に置換する。
s/(\S+)\s+(\S+)/$2 $1/;      # 最初の2つの単語を入れ換える。
```

変数展開はパターンをコンパイルする**前に**行われる。正規表現パーサには，変数展開後のパターン文字列が渡されるので，パターンに変数が含まれていたことは正規表現パーサからはわからない。それゆえ，変数の値に含まれるすべてのメタキャラクタは，（普通の文字ではなく）メタキャラクタとして認識されてしまう。変数に含まれるメタキャラクタを解釈しない（普通の文字として扱う）ためには，次のようにして英数字以外の文字すべての前に，逆スラッシュを挿入すればよい：

```
$string =~ s/(\W)/\\$1/g;
/a pattern containing a $string to be interpreted literally/;
```

[15] [訳注] substr演算子によって文字列から取り出したものも部分文字列と呼ぶので，混同しないように気をつけよう。たいていは文脈から区別できる。